<알고리즘 요약 정리본>

5장_1 탐색(이진탐색트리) (170410)

탐색에서 기본적으로 쓰이는 자료구조는 무엇이라고 생각하는가? 싱글 노드를 이용한 연결 리스트? 아니면 배열을 이용한 힙(Heap)? 그러나 연결 리스트보다 시간 복잡도가 줄어들면서 힙보다 구현하기 쉬운 엄청난 자료구조인 트리에 대해서 이번에는 공부를 해보는 기회를 가져보도록 하자.

0) 레코드, 키, 필드

우선 트리를 시작하기 앞서서 이러한 개념들이 무엇인지에 대해서 간략히 정리를 해보겠다. 어떠한 데이터들의 관계(혹은 계층 관계)를 표현하기 위해서 만든 자료구조가 바로 트리이다. 우리가 데이터베이스를 공부해봤을 때 이러한 개념들을 많이 봤을 것이다. 어떠한 것들이 있는지 살펴보도록 하자.

-> 레코드 : 우리가 어렸을 때 오락실에서 게임을 했을 때 게임오버가 나오면 최고 기록(Best Record)이 나온 것을 볼 수 있을 것이다. 이처럼 개체에 대해 수집된 모든 정보를 포함한 단위를 레코드라고 한다. 음료수의 레코드로 보면 브랜드 사, 용량, 유통기한 등등의 정보가 포함이 되겠다.

-> 필드 : 레코드에서 분류된 컬럼들에 대해서 각 정보를 나타낸 부분을 필드라고 한다. 아까의 사례로 브랜드 사를 보면 해타껀지 룻데껀지, 용량으로는 238밀으리인지 1.5리터인지, 유통기한으로서는 다음 달까진지 아니면 썩었는지 등등 세부 정보를 적은 개념을 바로 필드라고 한다.

-> 검색 키 : 다른 레코드와 중복되지 않도록 각 레코드를 대표로 하는 필드를 검색 키라고 한다. 음료수를 사례로 든다면 바코드가 될 수 있고, 핸드폰을 사례로 보면 전화번호 혹은 유심칩 번호로 살펴볼 수가 있겠다. 쉽게 말하자면 기본 키(Primary Key)와 같은 개념으로 보면 되겠다.

-> 검색 트리 : 검색 키를 통해서 레코드가 저장된 위치를 분별하기 위해 만든 자료구조를 검색 트리라고 한다. 우리가 살펴보는 이진 탐색 트리도 이러한 원리로 만들어졌다고 살펴보면 되겠다.

1) 트리(Tree)의 기본적 개념

트리는 우리가 생각하는 크리스마스트리나 한 몇 세기는 묵은 노수(老樹)를 생각해볼 수 있는데 근데 맞다. 나무를 뒤집어서 뿌리(root)에서 잎(leaf)까지 자료들이 퍼지는 모습을 보면서 만든 자료구조가 바로 트리이다. 그렇지만 이진 탐색 트리를 공부하기 앞서서 트리에 대해서 기본적인 개념에 대해서 알아두고 가면 약이 되시겠다. 어떠한 개념들이 있는지 살펴보도록 하자.

1-1) 이진 탐색 트리의 기본 개념

우리가 야구에서 삼진 아웃인 경우에 3번 아웃을 모두 해서 초에서 말로(아님 그 반대)로 바뀌는 경우를 살펴볼 수 있다. 이처럼 트리의 아들 원소의 개수를 2개로 제한을 하면서 만들어진 트리가 바로 이진 트리라고 할 수 있겠다. 이로서 데이터들의 크기를 통해 쉽게 비교를 하기 위해 만든 트리가 바로 이진 탐색 트리(Binary Search Tree)라고 볼 수 있다. 그럼 이진 탐색 트리의 기본적인 개념에 대해서는 아래와 같이 정리할 수 있겠다.

(참고) Key와 Value는...??

우리가 컬렉션(Collection) 클래스를 살펴보면 Key랑 Value가 있다는 사실을 알 수 있다. 그렇지만 우리가 대부분 살펴보는 이진 트리에는 int 형 데이터만 들어가기 때문에 Key를 정수형 값으로 살펴보는 시점에서 설명하겠다.

-> 각 노드에는 하나씩의 Key 값만 가질 수 있다.

이진 트리의 원칙은 바로 중복이란 것은 있을 수가 없는 일이다. 만일 같은 값이 들어오게 된다면 어느 값이 큰지, 어느 값이 작은지에 대해서 분별이 안 되기 때문에 하나씩의 Key 값만이 존재할 수밖에 없다.

-> 최상위 레벨에 루트가 존재하고, 자식 노드의 개수는 최대 2개로 한정되어 있다.

루트는 이진 탐색 트리가 포화 이진트리인 기준으로 중위 순회(inorder)를 하면 중앙값이 된다. 그리고 루트 노드부터 시작해서 자식들을 최대 2개씩 내려가게 되면서 그 자식들도 최대 2개씩 내려가게 되는 원리로 재귀적인 자료구조로 볼 수 있다. 루트 트리이든 간에 서브 트리든 간에 자식은 언제나 2개 이하로만 지정이 가능하다.

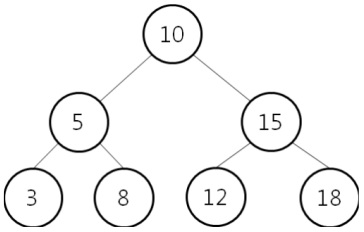
-> 자신보다 작은 값은 좌측 아들 노드로, 큰 값은 우측 아들 노드로 보낸다.

이 점도 마찬가지로 재귀적인 자료구조로 칭할 수가 있다. 어느 한 노드에 대해서 크기가 작으면 왼쪽으로 보내고 크면 오른쪽으로 보내는 것이 이진 탐색 트리의 원칙이다.

1-2) 트리의 종류

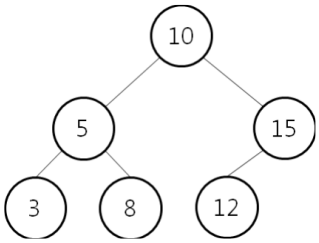
트리의 종류로서는 크게 3가지로서 나뉠 수 있다. 이를 그림으로 표현해서 쉽게 설명하겠다.

Case 1) 포화 이진트리



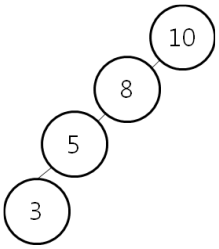
트리의 높이가 h라고 치면 포화 이진트리에 있어야 하는 원소의 수는 $2^{h+1}-1$이어야 포화 이진트리라고 할 수 있다. 지금 그림에 나와 있는 트리의 높이(혹은 레벨)은 2이기 때문에 7개의 원소가 모두 있어서 포화 이진트리라고 할 수 있다. 포화 이진트리인 경우에는 이진탐색을 하는데 있어서 좌측과 우측 모두 공평성이 주어져서 이진 탐색트리계의 완벽 조건이라고 볼 수 있다.

Case 2) 완전 이진트리



트리의 높이가 h라고 치면 트리 내의 원소의 수는 $2^h \le n \le 2^{h+1}-1$이어야 완전 이진트리라고 볼 수 있다. 왼쪽부터 모두는 아닐지라도 차례대로 채워진 트리가 바로 완전 이진트리로 볼 수 있겠다. 힙(Heap) 자료구조에서도 이러한 규칙이 성립이 되어야 만들어진 원리이기도 하고, 포화 이진트리도 완전 이진트리에 속한다. 그렇지만 왼쪽에 있는 값들에 대해서는 불공정하지만, 3번과 같은 트리에 비하면 아무것도 아니다.

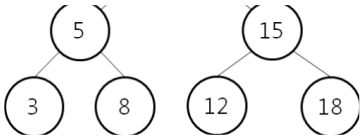
Case 3) 편향 이진트리



트리의 키 값들이 한 쪽으로 치우친 트리를 편향 이진트리라고 한다. 이는 쉽게 이야기해서 연결리스트랑 거스무리한 개념이 되어버리기 때문에 이러한 이진트리를 쓰게 되면 대부분 시간 복잡도는 O(n)이 나와버린다. 이러한 트리를 쓸 바에는 차라리 연결 리스트를 쓰는 것이 약이 되겠다.

1-3) 서브 트리

영어로 서브(sub)는 부분적이라는 뜻이다. 그래서 Case 1에서 살펴본 트리를 루트 노드를 없애버리고 두 동강 낸다면 이 녀석들이 바로 서브트리라고 볼 수 있겠다.



그리고 이러한 녀석들이 모두 모여 있는 개념을 바로 포레스트(forest)라고 볼 수 있다. 쉽게 이야기해서 우리가 과수원에서 사과를 따고 열매를 맺은 씨를 통해서 나무들을 심어서 다시 과수원을 이루는 원리라고 할까??

2) 이진 탐색 트리 알고리즘

이제 트리에 대한 간략한 개념에 대해서 알아봤으니 알고리즘에 접근해보도록 하자. 이진 트리 알고리즘은 힙 알고리즘이랑 비교를 해보면 따로 따로 참조하는 값에 대해서도 생각을 해야 되기 때문에 Child Node(자식 노드)를 이용해서 접근을 해야겠다. 트리에 어떠한 값을 넣느냐에 따라서 포화 이진트리가 나올 수도 있고 아니면 편향 이진 트리도 나올 수 있기 때문에 <u>같은 데이터들(그니깐 들어갈 데이터들과 개수는 같은데 순서를 바꾸게 되면)에 대해서도 고려를 해줘야 하니 알아두도록 하자.</u>

2-0) 이진 탐색 트리의 데이터 구조는?

```
public class TreeTest{
    static class Node{
      int value;
      Node left, right;
    }
    public Node(int value){
      this.value=value;
      this.left=this.right=null;
    }
    // *
}
```

우선 이진 탐색 트리를 시작하기 앞서서 내부 클래스의 개념으로 작성을 하였다. 내부 클래스는 아시다시피 public class 내부에 있는 메소드 내부에서만 쓸 수 있는 클래스의 개념으로 생각하면 되겠다. 쉽게 말해서 집 안에 있는 강아지를 학교에 모시고 오기에는 벅차니깐... 물론 일반 클래스를 만들어서 해도 되는데 여기선 public 클래스(기본 클래스) 내부에서만 돌아가게 함으로서 보안성을 강화하기 위해서 작성하였다고 생각해보자.

그건 둘 째 치고, Node라는 녀석은 이진 탐색 트리에 쓰이는 키의 값과 자식들을 보관하는 객체 중 일부인데 left와 right는 작은 값이면 left로 보내고, 큰 값이면 right로 보내게 된다. 여기 에스터링크(*) 이후로 작성되는 이진 탐색 트리에 관련된 알고리즘에 대해 정리해보도록 하자.

2-1) 이진 탐색 트리에서의 탐색

```
public Node search(Node node, int value){
   if(node.null || node.key==value) return node; // 1
  if(value<node.key) return search(node.left, value); // 2
  else return search(node.right, value); // 3
}
```

이진 탐색 트리를 살펴보는 기준은 바로 큰가 아니면 작은가가 바로 핵심이 되겠다. 만일 데이터의 크기가 작으면 왼쪽을 족치면 되고, 반대로 크다면 오른쪽을 족치면 되는 원리가 바로 이진 탐색 트리이다. 이진 탐색 트리는 재귀적 구조로 되어 있기 때문에 어떤 자식으로 내려가도 그 원리는 그대로 적용이 되어서 분별하기 쉽다.

1 : node가 null인 경우에도 어자피 null이란 녀석도 반환을 해야 돼서 이렇게 작성하였지만, 또한 value과 같은 값을 찾은 경우에도 반환을 해야 돼서 node를 반환하는 기준에서 or 관계 연산자로 끝내는 방안으로 이용되었다 보면 된다.

2 : 만일 현재 노드의 키보다 작은 경우에는 당연히 왼쪽 자식부터 시작하는 서브트리를 족치러 가야 된다. 그래서 좌측 서브트리에 관련되어 재귀적 호출을 해준다.

3 : 반면 키보다 큰 경우에는 오른쪽 자식부터 시작하는 서브트리를 족쳐야 돼서 우측 서브트리에 관련된 재귀적 호출을 한다.

이처럼 이진 탐색 트리의 탐색 알고리즘의 시간 복잡도는 일반적으로 $O(\log n)$이 된다. 하지만 만의 하나

의 일로 편향 이진트리와 같은 저질 이진 트리가 나오게 되면 O(n)이 되어버려서 일반적인 시간 복잡도도 무의미하게 된다.

2-2) 이진 탐색 트리 삽입 알고리즘

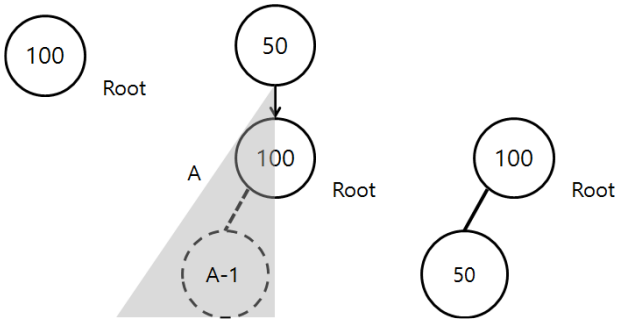
```
public void add(int value){
    if(value<this.value){ // A
      if(this.left==null) // A-1
        this.left=new Node(value);
      else left.add(value); // A-2
    } else if(value>this.value) { // B
      if(this.right==null) // B-1
        this.right=new Node(value);
      else right.add(value); // B-2
    }
}
```

삽입 알고리즘은 아직까지 그렇게 어려운 편이 아니다. 그렇지만 어떻게 작동을 하는지에 대해서 알아볼 필요가 있겠다. 아래와 같은 값들을 추가하면서 원리를 익혀보도록 하자.

<작동 사례> 이미 root 값에는 100이 저장되어 있다고 가정을 하겠다.

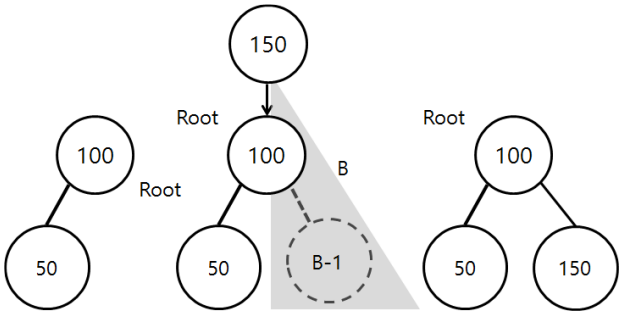
50, 150, 75, 111까지 추가하는 알고리즘에 대해 설명하겠다.

1> 100에 50을 삽입 해보면...



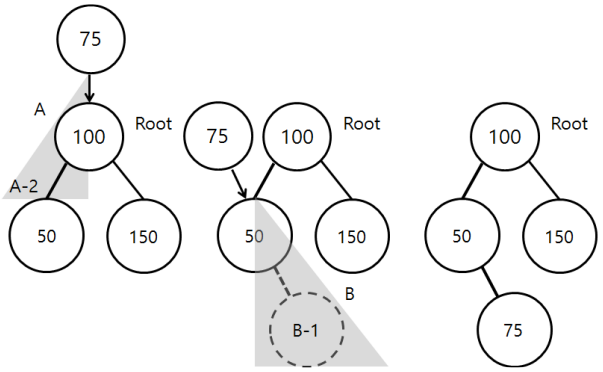
일단 50은 당연히 100보다 작기 때문에 100의 좌측 자식을 쳐다볼 수 밖이다. 그래서 A의 문장이 이루어지면서 100의 좌측 노드가 null이기 때문에 왼쪽에 50을 저장하는 것으로 종결한다.

2> 그 다음 150을 삽입 해보면...



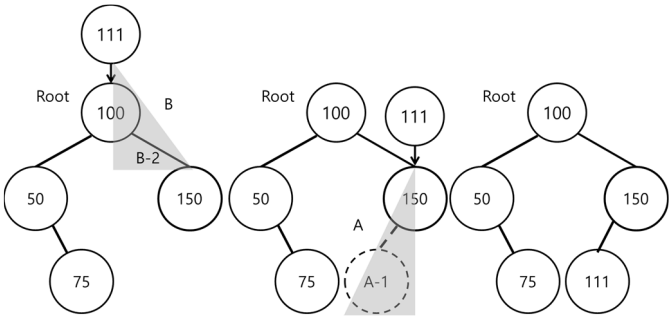
그 다음 150이란 값을 삽입하는 경우에는 일단 150이 100보다 크기 때문에 100의 우측 자식을 노리고 있다. 그래서 B 문장으로 넘어가게 되면서 또한 100의 우측 자식이 없기 때문에 바로 삽입을 하면서 마지막처럼 완성을 하게 된다.

3> 그 다음 75를 삽입하게 된다면...



우선 75를 삽입하는 과정을 살펴본다면, A의 문장처럼 이미 100(root)보다 작기 때문에 왼쪽으로 가게 된다. 하지만 이미 root 좌측 자식은 null이 아니라서 A-2로 넘어가게 되면서 50에 대한 서브 트리를 촉치게 된다. 그렇지만 75는 50보다 한참 크기 때문에 B 문장으로 넘어가게 되면서, 50의 우측 자식 노드가 null이라서 B-1로 넘어가게 되면서 B-1 문장을 통해서 50의 우측 자식 노드에 75를 삽입하게 됨으로서 종결된다.

4> 그 다음 111를 삽입하게 된다면...



111은 일단 100보다 크기 때문에 B 문장을 실행하게 된다. 그렇지만 100의 우측 자식 노드는 이미 존재하기 때문에 B-2 문장을 통해서 150부터 시작하는 서브 트리에 대해 촉치게 된다. 그 뒤로 111은 150보다 작기 때문에 A의 문장을 실행하게 되면서 150의 좌측 자식 노드의 값은 null이기 때문에 A-1 자리에 111를 채워줌으로서 최종적인 모양이 나오고 종결된다.

5> 만일 있는 값들을 다시 삽입한다면...??

일단 A와 B를 거치게 된다면 어떻게든 재귀를 할 것이다. 그렇지면 value==this.value란 조건이 따로 없기 때문에 이는 무시하게 되면서 아무런 값들이 추가되지 않고 결국엔 삽입이 되지 않는다. **<u>본래 이진 탐색 트리의 원칙이 중복된 값이 추가되는 것을 방지해야 하기 때문이다.</u>**

2-3) 이진 탐색 트리의 중위 순회 알고리즘

우리가 이진 트리에 대해서 순회하는 알고리즘을 통해서 알고리즘을 분석한 정리를 봤을 것이다. 중위 순회에 대해서는 시간 복잡도가 이진 탐색 트리의 노드 수이기 때문에 O(n)이 나오게 된다. 이는 아래처럼 정리할 수 있겠다.

```
public void print(){
    if(left!=null) left.print(); // 1
    System.out.printf("%d ", this.value); // @
    if(right!=null) right.print(); // 2
}
```

우리는 이걸 왜 중위 순회라고 할까? 매우 쉽게 생각하는 방법으로는 @의 위치가 1번이면 전위 순회, 2번이면 후위 순회로 생각할 수 있다.(물론 전위 순회는 1번 문장은 @로 가야 되고, 후위 순회이면 2번 문장이 @로 가야 된다.) 하지만 이진 탐색 트리를 중위 순회를 하게 된다면 값이 작은 값부터 시작해서 큰 값으로 해서 왼쪽↗중앙↘오른쪽 순서대로 순회를 하니 자동적으로 오름차순 정렬이 된다. 이것이 바로 이진 탐색 트리의 숨겨진 매력으로 살펴보면 되겠다.

2-4) 이진 탐색 트리 삭제 알고리즘

이진 탐색 트리 삭제 알고리즘은 생각보다 어렵다기보다 경우를 잘 따지면 오히려 괜찮은 알고리즘으로 살펴볼 수 있다. 삭제를 하는데 있어서 다음과 같은 경우들을 고려해야겠다.

-> 삭제하려는 노드가 자식이 아무 것도 없을 때

-> 삭제하려는 노드가 자식이 하나일 때(왼쪽 혹은 오른쪽에 자식이 하나일 때)

-> 삭제하려는 노드가 자식이 모두 있을 때(왼쪽 서브트리에서 최댓값을 가져오는 방법, 오른쪽 서브트리에서 최솟값을 가져오는 방법)

이러한 경우들을 기준으로 어떠한 알고리즘으로 작성이 되는지 차례대로 알아보도록 하자.

Case 01> 삭제하려는 노드가 자식이 없을 때

```
if(parent.left==this)
    parent.left=null;
else
    parent.right=null;
```

Case 02> 삭제하려는 노드가 자식이 하나인 경우

```
if(this.left!=null) child=this.left;
else child=this.right;
if(parent.left==this) parent.left=child;
else parent.right=child;
```

Case 03> 삭제하려는 노드가 자식을 모두 갖춘 경우

```
int temp=right.getLeftMostValue();
this.value=temp;
right.remove(value, this);
```

생각보다 이에 대해서 융합을 하는 모습을 보여야 되는데 감이 잡히지 않을 것이다. 일단 Case 01, 02를 종합시켜보면 아래와 같이 작성할 수 있을 것이다.

```
Node child;
if(this.left!=null) child=this.left; // 가
  else child=this.right;
if(parent.left==this) parent.left=child; // 나
  else parent.right=child;
```

가 : 현재 노드의 왼쪽 자식이 하나 있다면 가 문장을 실행하게 될 것이다. 그렇지만 오른쪽 자식이 있거나 자식이 아예 없으면 child에 right를 저장을 해두든 null을 저장을 해두게 된다.

나 : 부모의 왼쪽 자식이랑 현재 노드랑 같으면 현재 노드의 자식노드로 바꿔버리는 문장이 바로 나 이다. 아니면 parent의 right 노드가 같게 된다면 현재 노드의 자식으로 바꾸게 된다. 방금 언급했던 child에는 left가 저장될 수도 있고, right도 저장될 수도 있고, null도 저장이 될 수 있기 때문에 이처럼 종합이 가능하다.

이 경우들에 대한 알고리즘을 종합해서 아래와 같은 알고리즘을 살펴보도록 하자.

```
public void remove(int value, Node parent){
    if(value<this.value){ // A
      if(left!=null) left.remove(value, this);
    } else if(value>this.value){ // B
      if(right!=null) right.remove(value, this);
    } else{
      if(left!=null && right!=null){ // C
        int temp=right.getLeftMostValue();
        this.value=temp;
        right.remove(temp, this);
      } else { // D
        Node child = (left!=null) ? left:right;
        if(parent.left==this) parent.left=child;
        else parent.right=child;
      }
    }
}
```

```
public int getLeftMostValue(){
    if(this.left!=null)
      return left.getLeftMostValue();
    return this.value;
}
```

알고리즘이 쿽 정렬만큼의 길이인데 처음에 봐도 무슨 개소리인지 잘 모를 것이다. 그렇지만 아까 언급했던 경우들을 사례로서 설명을 들어 가보자.

A : 우선 value랑 현재 가지고 있는 노드의 키 값이랑 비교를 해서 작은 경우에 왼쪽 서브트리를 족치는 원리로 생각하면 되겠다.

B : 이는 value랑 현재 가지고 있는 노드의 키 값이랑 비교를 해서 큰 경우에 오른쪽 서브트리를 족치는 원리로 생각하면 되겠다.

C ; 이는 값들을 서로 비교하면서 만일 자식들의 수가 2개인 노드에 대해서 삭제 작업을 들어가게 된다면 temp의 값을 오른쪽 서브트리에서 최솟값을 골라내면서 그 값으로 대체하는 과정으로 살펴보면 되겠다.

D : 이는 자식이 하나이거나 자식이 없는 경우에 돌아가는 문장으로 살펴보면 되겠다. 자식이 하나여도 그 자식에게 넘겨주거나 어자피 자식이 없는 노드는 알아서 null을 대입해서 자기가 잘려 나가는 원리로 생각하면 되겠다.

A에서 D까지의 경우를 말로만 설명했는데 이는 그림으로 보면서 정리를 하는 것이 선택이 아닌 필수이다. 그림을 이용해서 이 알고리즘을 이해하도록 하자.

이는 한글 파일로 작성하기에는 무리가 있어서 PPT 자료로 따로 첨부하였으니 **170410_5_1장_이진트리_삭제_알고리즘.ppt 파일을 참고**하면 되겠다.

2-5) 이진 탐색 트리 포함 알고리즘

우리가 이진 탐색 트리에 존재하는 값들에 대해서 알고 싶을 경우가 있을 것이다. 마치 Collection 클래스에 있는 contains 인터페이스같은 역할처럼 말이다. 그래서 이를 구현한 문장은 아래와 같이 작성할 수 있겠다.

```
public boolean contains(int value){
  if(value<this.value) // A
    return (left!=null) && left.contains(value);
  else if(value>this.value) // B
 return (right!=null) && right.contains(value);
  return true; // C
}
```

A : 만일 입력한 value의 값이 작게 된다면 왼쪽 서브트리를 통해서 값이 포함되어 있는지에 대해서 재귀 호출을 하는 방안이 있다. 만일 value가 존재하지 않는 경우에는 애석하게도 left가 null이 분명 나올 것이다. 본래 이 문장은 아래와 같이 풀어서 썼다고 생각을 해보면 쉽겠다.

```
if(value<this.value){
    if(left==null) return false;
    else left.contains(value);
}
```

B : 이도 마찬가지로 value의 값이 크게 된다면 오른쪽 서브트리를 통해서 값이 포함되어 있는지에 대해서 재귀 호출을 하게 된다. 그래서 존재하지 않으면 right가 null이 분명 나올 것이다. 이도 마찬가지로 이렇게 풀어쓸 수 있겠다.

```
else if(value>this.value){
    if(right==null) return false;
    else right.contains(value);
}
```

C : 만일 값이 작지도 않고, 크지도 않고 결국에는 같은 값이 나오게 된다. 그래서 결국 value라는 녀석이 존재한다는 뜻으로서 true를 반환을 하고 종결하게 된다.

참고로 이 알고리즘에 대해서도 ppt 자료를 이용해서 보기 쉽게 정리해뒀으니 공부하는데 있어서 참조하면 도움이 되겠다.

<참고> 현재 가지고 있는 remove 알고리즘으로는 root 노드를 지우는데 있어서 무리가 있다. 그래서 아래와 같이 Node 내부 클래스와 같은 위치에 있는 내부 클래스를 만들어서 개선을 하는 방법을 생각해보겠다.

```
static class BinaryTree {
    Node dummy;
    public BinaryTree() {
    dummy = new Node(Integer.MAX_VALUE);
    } // 1
    public void add(int value) {
        dummy.add(value);
    }
    public void print() {
    if (dummy.left != null) dummy.left.print();
        System.out.println();
    }
    public boolean contains(int value) {
        return dummy.left != null &&
dummy.left.contains(value);
    }
    public void remove(int value) {
        if (dummy.left != null)
         dummy.left.remove(value, dummy);
    } // 2
}
```

1 : 여기서 dummy라는 뜻은 오로지 장식용으로 두겠다는 뜻이다. 쉽게 생각해보면 돈이 많다고 자랑하기 위해 돈더미를 보여주는 것을 연상하면 쉽겠다. 그럼 이 dummy는 어떤 생각으로 만들어뒀을까? 바로 Integer.MAX_VALUE(이게 약 21억인데 $2^{31}-1$ 정도 될 것이다...)라는 값을 통해서 int 형의 끝장판을 저장해둠으로서 이의 왼쪽 서브트리는 **int 범위 이내에서 트리를 만들 수 있도록 형성을 한 원리로 생각**하면 되겠다. 그럼 int 범위 이내에 만든 root 는 결국 dummy 란 어마한 애가 맡고 있어서 root도 삭제할 수 있게끔 만들어진다.

2 : 여기서 int 형 범위 이내에 만든 서브트리는 어디에 저장될까? 당연히 MAX_VALUE보다 작은 왼쪽에 저장될 수밖에 없다. 그래서 왼쪽 서브트리의 값들이 존재하는지에 대한 확인절차가 필요로 하다. add 메소드는 int 범위 이내의 값을 입력하게 된다면 어자피 add 메소드 그대로 돌아갈 것이고, print도 마찬가지로 왼쪽 서브트리가 존재한다면 그 범위만 출력을 하게 되니깐 왼쪽 서브트리만 고생을 하게 된다. 이처럼 remove 메소드를 작성해주게 된다면 int 형 범위 이내에 만든 서브트리에서 root를 삭제하는데 있어서 큰 문제가 없게 됨으로서 정상적으로 삭제가 된다.

하지만 이렇게 작성을 하게 된다면 큰 데이터의 값에 대해서 용량은 비록 똑같지만 데이터에 대한 하나의 값이 늘어나게 되어 int 범위 내에 있는 서브트리가 없더라도 공간 낭비에 대해선 고려를 못한다. 그렇지만 없는 것보단 낫다고 생각을 할 수가 있는데 root 값도 언젠가는 지워야 한다. 이러한 방안에 대해서도 생각을 해봄으로서 강의노트를 참조하면 공부하는데 도움이 될 것이다.